A New MPI Implementation for Cray SHMEM

Ron Brightwell

Scalable Computing Systems
Sandia National Laboratories*
P.O. Box 5800
Albuquerque, NM 87185-1110
rbbrigh@sandia.gov

Abstract. Previous implementations of MPICH using the Cray SHMEM interface existed for the Cray T3 series of machines, but these implementations were abandoned after the T3 series was discontinued. However, support for the Cray SHMEM programming interface has continued on other platforms, including commodity clusters built using the Quadrics QsNet network. In this paper, we describe a design for MPI that overcomes some of the limitations of the previous implementations. We compare the performance of the SHMEM MPI implementation with the native implementation for Quadrics QsNet. Results show that our implementation is faster for certain message sizes for some micro-benchmarks.

1 Introduction

The Cray SHMEM [1] network programming interface provides very efficient remote memory read and write semantics that can be used to implement MPI. Previously, the SHMEM interface was only available on the Cray T3D and T3E machines and implementations of MPICH using SHMEM were developed specifically for those two platforms [2, 3]. Recently, SHMEM has been supported on other platforms as well, including machines from SGI, Inc., Cray, Inc., and clusters interconnected with the Quadrics network [4].

This paper describes our motivation for this work and presents a design that overcomes some of the limitations of these previous implementations. We compare the performance of the SHMEM MPI implementation with the native implementation for Quadrics QsNet. Micro-benchmark results show that the latency performance of the SHMEM implementation is faster for a range of small messages, while the bandwidth performance is comparable for a range of large messages.

The rest of this paper is organized as follows. The next section describes how this work relates to other published research. Section 3 discusses the motivations for this work. The design of our MPI implementation is presented in Section 4, which is followed by performance results and analysis in Section 5. Section 6 outlines possible future work.

* Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

2 Related Work

The design and implementation of MPICH for SHMEM described in this paper is a continuation of previous work for the Cray T3D [2], which was subsequently ported to the Cray T3E [3]. This new implementation is less complex than the previous implementations, due to some extended features that are provided in the Quadrics implementation of SHMEM. In particular, the T3 machines did not support arbitrarily aligned transfers and required explicit cache-management routines to be used. The SHMEM implementation for Quadrics provides the ability to transfer data without any alignment or length restriction, and the PCI-based network eliminates the need to explicitly manage the data cache.

Recently, some of the techniques that were employed in these earlier MPI/SHMEM implementations have been used to support MPI implementations for InfiniBand. For example, the method of using a fixed set of buffers at known locations for handling incoming messages, which has been referred to as *persistent buffer association*, has been used in the MVAPICH implementation [5]. The similarities in the SHMEM interface and the remote DMA (RDMA) operations provided by the VAPI interface are largely what motivated this updated MPI/SHMEM implementation. We discuss more details and further motivations in the next section.

3 Motivation

The native Quadrics implementation of MPI uses the Tports interface to offload all MPI matching and queue traversal functionality to the network interface processor. In most cases, this is an ideal approach. However, there are some cases where matching and queue traversal can be more efficiently handled by the host processor, as is done in the SHMEM implementation. We have used this implementation along with the Tports implementation for extensive comparisons that quantify the benefits of independent progress, overlap, and offload for applications [6–8], using an identical hardware environment.

The MPI/SHMEM implementation also has some features that are not provided by the MPI/Tports implementation. For example, data transfers are explicitly acknowledged in the MPI/SHMEM implementation, while they are not for the MPI/Tports implementation. This approach may be beneficial for applications where load imbalance causes messages to be produced faster than they can be consumed. The Tports interface handles buffering of unexpected messages implicitly, so unexpected messages that arrive from the network are deposited into buffers that are allocated and managed within the Tports library. This space never really becomes exhausted, as the Tports library will keep allocating more memory, relying on virtual memory support to provide more. In some cases, a protocol that throttles the sender is more appropriate. The MPI/SHMEM implementation provides this throttling since the buffers used for MPI messages are explicitly managed by both the sender and the receiver.

Finally, the SHMEM interface has capabilities that are very similar to those provided by the current generation of networking technology that supports RDMA operations, such as InfiniBand. The distributed shared memory model of SHMEM avoids the

need for the initiator and the target of a put or get operation to explicitly exchange information in order to begin a transfer, but the semantics of the transfer and mechanisms used to recognize the completion of transfers are very similar. This is especially true for some of the MPI implementations for InfiniBand that use RDMA operations [5, 9, 10]. We intend to use the SHMEM interface to analyze characteristics, such as strategies for efficiently polling for incoming messages, that may be beneficial to RDMA-based implementations of MPI.

4 Design and Implementation

4.1 Basic Data Transfer Mechanism

Here we describe our basic scheme for message passing using the SHMEM remote memory write (put) and remote memory read (get) operations. A point-to-point transfer between two processes can be thought of as a channel. The sender fills in the MPI envelope information and data in a packet and uses the remote write operation to transfer this packet to the receiver. On the receive side, the receiver recognizes the appearance of a packet and handles it appropriately.

Figure 1 illustrates the contents of a packet. The largest area of a packet is for user data. For our Quadrics implementation, the size of the data field in a packet was 16 KB. Following the data is the MPI envelope information: the context id of the sending MPI communicator, MPI tag, and the length of the message. We also include the local source rank within the communicator as an optimization to avoid a table lookup at the receiver. In addition to this information, a packet also includes two fields, `Send Start` and `Send Complete` that are used for the long message and synchronous message protocols. We will describe their use below in Section 4.2. Finally, the last field in the packet header is the `status` field, which is used to signal the arrival and validity of a packet.

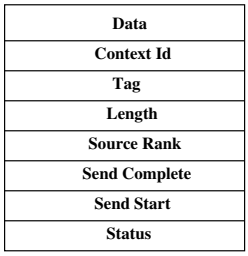

Fig. 1. Packet Structure

There are a few important distinctions about the way in which a packet is constructed. First, the status field must be the last field in the packet, since it signifies the arrival and validity of a packet. Fixing the location of the status within the structure avoids having to use more complex techniques, such as those described in [5], to decipher when a complete packet has arrived. The receiver must only poll on a single memory location to determine packet arrival.

There are two possible strategies for sending a packet. First, two individual put operations could be used. The first put would be used to transfer the user data portion of a packet, and the second put would transfer the MPI envelope information. However, this essentially doubles the network latency performance of a single MPI send operation. For implementations of SHMEM where there is no ordering guarantee for successive put operations, an additional call to a synchronization function, such as `shmem_fence` may be needed to insure that the second put operation completes after the first. The second strategy copies the user data into a contiguous packet and uses a single put operation to transfer both the data and the MPI envelope. This is the strategy that we have used in our implementation. Since the length of the data portion of a packet is variable, the data is copied into a packet at an offset from the end of the buffer rather than from the beginning. This means that the start of a packet varies with the size of the data, but the end of the packet is always fixed. This way, a packet can be transmitted using a single put operation.

Since the target of a put operation must be known in advance, it is easiest to allocate send packets and receive packets that are *symmetric*, or mapped to the same virtual address location in each process. Our current implementation does this by using arrays that are statically declared such that there are `N` packets for sending messages to each rank and `N` corresponding receive packets for each rank. Packets for each rank are managed in a ring, and a counter is maintained for each send and receive packet for each rank to indicate the location of the next free packet.

In order for rank 0 to send a packet to rank 1, rank 0 checks the status field of the current send packet. If the status is set, this indicates that the corresponding receive packet at the destination has not yet been processed. At this point, the sender looks for incoming messages to process while waiting for the status to be cleared. When the status is cleared, the sender fills in the MPI envelope information, copies the data into the packet, sets the status field, and uses a put operation to write the packet to the corresponding receive packet at the destination. The sender then increments the counter to the next send packet for rank 1.

To receive a packet, rank 0 checks the status flag of the current receive packets for all ranks by looping through the receive packets array. Eventually, it recognizes that rank 0 has written a new packet into its current receive packet slot. It examines the contents of the MPI envelope and determines if this message is expected or unexpected and whether it is a short or long message. Once processing of this packet is complete, it clears the local status flag of the receive packet that was just processed, uses a put operation to clear the status flag of the send packet at the sender, and increments the current location of the receive packet for rank 0.

4.2 Protocols

Our implementation employs a traditional two-level protocol to optimize latency for short messages and bandwidth for long messages. Short messages consist of a single packet. Once the packet is sent, a short message is complete. For long messages, the data portion of a packet is not used. The MPI envelope information is filled in, the `Send Start` portion of the packet is set to the location of the buffer to be sent, and the `Send Complete` field is set to the location of the completion flag inside the MPI send

request handle. When the packet is received and a matching receive has been posted, the receiver uses a remote read operation to read the send buffer. Once the get operation is complete, it uses a remote write operation to set the value of the completion flag in the send handle to notify the sender that the transfer has finished. The `Send Complete` field is also used to implement an acknowledgment for short synchronous send mode transfers.

4.3 Unexpected Messages

When a packet is received, the posted receive queue is checked for a match. If no match is found, a receive handle is allocated and the contents of the packet are copied to the handle. If the message is short, a temporary buffer is allocated and the data portion of the packet is copied into it. Once a matching receive is posted, the contents of this buffer are copied into the user buffer and the temporary buffer is freed.

This implementation is very similar to the implementation for the T3E, with a few optimizations. First, the T3E implementation used only one send packet and one receive packet for each destination. We've enhanced this using a ring of packets to allow for several outstanding transfers between a pair of nodes. The T3D implementation used two put operations for each send – one for the MPI envelope information and one for the data. Because of the low latency of the put operation, copying the user data into a packet incurred more overhead than sending the data using a separate remote write operation. However, since the T3E supported adaptive routing, successive remote writes were not guaranteed to arrive in the order they were initiated. As such, the T3E implementation used a memory copy and a single remote write operation, as does our implementation for Quadrics.

4.4 Limitations

While this implementation demonstrates good performance for micro-benchmarks, it does have some drawbacks that may impact scalability, performance, and usability. Since send and receive packets are allocated using host memory, the amount of memory required scales linearly with the number of processes in the job. If we have 8 buffers each of size 16 KB, each rank requires 128 KB of memory. For a 1024 process job, this amounts to 128 MB of memory just for the packets alone. The current implementation does not allocate this memory dynamically, mostly because the Quadrics implementation does not support the Cray `shmalloc` function for obtaining symmetric memory from a heap[1] The current library is compiled to support a maximum of 128 processes, so a significant amount of memory is wasted for jobs with fewer processes.

In addition to memory usage, the time required to look for incoming packets increases with the number of processes as well. Polling memory locations is not a very efficient way to recognize incoming transfers. We plan to explore several different strategies for polling and analyze their impact on performance. This limitation is of particular interest because it is also an issue for implementations of MPI for InfiniBand.

[1] There is an equivalent function call in the lower-level Elan libraries that could be used.

The rendezvous strategy that we employ for long messages does not support independent progress, so the opportunity for significantly overlapping computation and communication for large transfers is lost. Our implementation could be enhanced by using a user-level thread to insure that outstanding communication operations make progress independent of the application making MPI library calls.

Our implementation also does not use the non-blocking versions of the put and get operations. While these calls are listed the Quadrics documentation, they are not supported on any of the platforms to which we have access.

Finally, our implementation only supports the SPMD model of parallel programming. This is a limitation imposed by the SHMEM model that does not allow using different executables files in the same MPI job.

5 Performance

5.1 Platform

The machine used for our experiments is a 32-node cluster at Los Alamos National Laboratory. Each node in the cluster contains two 1 GHz Intel Itanium-2 processors, 2 GB of main memory, and two Quadrics QsNet (ELAN-3) network interface cards. The nodes were running a patched version of the Linux 2.4.21 kernel. We used version 1.24-27 of the QsNet MPI implementation and version 1.4.12-1 of the QsNet libraries that contained the Cray SHMEM compatibility library. The SHMEM MPI library is a port of MPICH version 1.2.5. All applications were compiled using Version 7.1 Build 20031106 of the Intel compiler suite. All of our experiments were run using only one process per node using only one network interface, and all results were gathered on a dedicated machine.

5.2 Results

Figure 2 shows the latency performance for the MPI/SHMEM implementation and the vendor-supplied MPI/Tports implementation. The zero-length latency is 5.6 μsecfor MPI/Tports and 6.6 μsecfor MPI/SHMEM. At 256 bytes, the SHMEM implementation begins to outperform the Tports implementation. This trend continues until the size of the message is 4 KB.

Figure 3 shows the bandwidth performance of the two implementations. From 10 KB to 200 KB, there is a difference of a little more than 20 MB/s in favor of the Tports implementation. This difference is largely attributable to the ability of the Tports implementation to avoid involving the host processor in large data transfers. In contrast, the SHMEM implementation must rely on the application process to initiate a remote memory read operation. At a message size of 100 KB, the margin between the two implementations has decreased to less than 1 MB/s, and only a minimal difference can be perceived for messages beyond that point.

Figure 4 illustrates how the posted receive queue can affect latency performance. For this measurement, there are 10 requests in the posted receive queue, and the percentage of the queue that must be traversed in order to receive a zero-length message

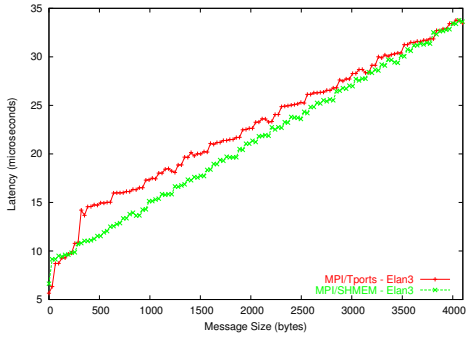

Fig. 2. Message latency for MPI/SHMEM and MPI/Tports

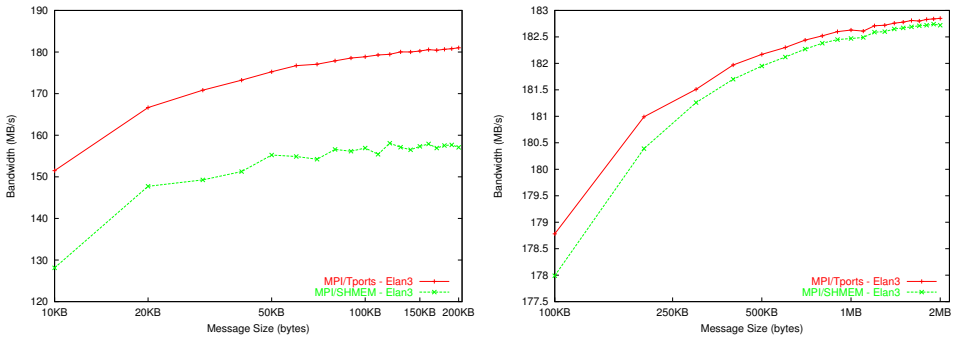

Fig. 3. Medium (a) and Long (b) Message bandwidth for MPI/SHMEM and MPI/Tports

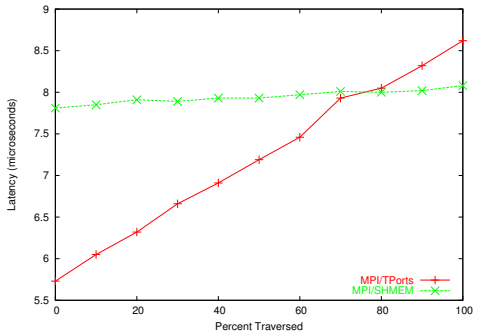

Fig. 4. Pre-posted message latency for MPI/SHMEM and MPI/Tports

is varied. The latency of the SHMEM implementation is less than the Tports implementation at 8 pre-posted receives. At this point, using the host processor rather than the network interface processor to perform MPI communicator and tag matching operations becomes more efficient.

6 Future Work

One of the important areas we intend to pursue with this work is the effect of memory polling strategies on application performance. For applications that have relatively few sources of incoming messages, we expect to be able to develop strategies that allow incoming messages to be discovered more quickly than simply polling all possible incoming message locations. We also intend to explore these strategies for other RDMA-based implementations of MPI, including InfiniBand.

There is much work that could be done to improve the performance of our MPI/SHMEM implementation. Collective operations are currently layered on top of MPI point-to-point functions, so work could be done to leverage the SHMEM collective routines that are available. SHMEM also has efficient support for non-contiguous transfers, so there might be some benefit for using these functions to handle non-contiguous MPI data types. Additionally, there may be some benefit for using the SHMEM interface for implementing the MPI-2 one-sided operations, since both two-sided and one-sided operations could be handled by the same transport interface.

References

1. Cray Research, Inc.: SHMEM Technical Note for C, SG-2516 2.3. (1994)
2. Brightwell, R., Skjellum, A.: MPICH on the T3D: A case study of high performance message passing. In: Proceedings of the Second MPI Developers' and Users' Conference. (1996)
3. Hebert, L.S., Seefeld, W.G., Skjellum, A.: MPICH on the Cray T3E. In: Proceedings of the Third MPI Developers' and Users' Conference. (1999)
4. Petrini, F., chun Feng, W., Hoisie, A., Coll, S., Frachtenberg, E.: The Quadrics network: High-performance clustering technology. IEEE Micro **22** (2002) 46–57
5. Liu, J., Wu, J., Kini, S.P., Wyckoff, P., Panda, D.K.: High performance RDMA-based MPI implementation over InfiniBand. In: Proceedings of the 2003 International Conference on Supercomputing (ICS-03), New York, ACM Press (2003) 295–304
6. Brightwell, R., Underwood, K.D.: An analysis of the impact of overlap and independent progress for MPI. In: Proceedings of the 2004 International Conference on Supercomputing, St. Malo, France (2004)
7. Brightwell, R., Underwood, K.D.: An analysis of the impact of MPI overlap and independent progress. In: 2004 International Conference on Supercomputing. (2004)
8. Underwood, K.D., Brightwell, R.: The impact of MPI queue usage on latency. In: Proceedings of the 2004 International Conference on Parallel Processing. (2004)
9. Liu, J., Jiang, W., Wyckoff, P., Panda, D.K., Ashton, D., Buntinas, D., Gropp, W., Toonen, B.: Design and implementation of MPICH2 over InfiniBand with RDMA support. In: Proceedings of the 2004 International Parallel and Distributed Processing Symposium. (2004)
10. Rehm, W., Grabner, R., Mietke, F., Mehlan, T., Siebert, C.: An MPICH2 channel device implementation over VAPI on InfiniBand. In: Proceedings of the 2004 Workshop on Communication Architecture for Clusters. (2004)